Local Tolerance to Unbounded Byzantine Faults

Mikhail Nesterenko * Anish Arora †

Abstract

An ideal approach to deal with faults in large-scale distributed systems is to contain the effects of faults as locally as is possible and, additionally, to ensure some type of tolerance within each fault-affected locality. Existing results using this approach accommodate only limited faults (such as crashes) or assume that fault occurrence is bounded in space and/or time. In this paper, we define and explore possibility/impossibility of local tolerance with respect to arbitrary faults (such as Byzantine faults) whose occurrence may be unbounded in space and in time. Our positive results include programs for graph coloring and dining philosophers, with proofs that the size of their tolerance locality is optimal. The type of tolerance achieved within fault-affected localities is self-stabilization. That is, starting from an arbitrary state of the distributed system, each non-faulty process eventually reaches a state from where it behaves correctly as long as the only faults that occur henceforth (regardless of their number) are outside the locality of this process.

Keywords: fault-containment, fault-tolerance, self-stabilization, Byzantine faults, dining philosophers

Contact Author: Mikhail Nesterenko, `mikhail@cs.kent.edu`

---

*Department of Computer Science, Kent State University, Kent, OH 44242 USA, `mikhail@cs.kent.edu`. This research was supported in part by DARPA contract OSU-RF #F33615-01-C-1901.

†Department of Computer and Information Science, Ohio State University, Columbus, OH 43210 USA, `anish@cis.ohio-state.edu`. This research was supported in part by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, Ohio State University Fellowship, and a 2002-2003 grant from Microsoft Research.

1 Introduction

Achieving reliability in large-scale distributed systems poses unique challenges. As system size grows, the incidence and relative duration of faults increases. Faults interact with system programs to create failure scenarios that are hard to anticipate. Faulty information can spread through the system. Such challenges motivate the need for fault tolerance techniques that are scalable in the sense that they achieve tolerance effectively despite growth in system size.

Together with challenges, large scale in distributed systems also provides opportunities for effective fault tolerance. As system size grows, the size of the regions where faults occur may not grow as fast. We may therefore assume that any region where faults have occurred is surrounded by nodes where faults have not occurred, thereby making it possible to deal with faults in a local manner. For scalable fault tolerance, then, an ideal goal is (I) to contain the effects of faults as locally as is possible and (II) to additionally ensure that a desired type of tolerance holds within the locality of containment. By way of example of (II), one may enforce safety within the locality or recover the locality so as to resume correct behavior.

Although the ideas of fault containment and local fault tolerance have been studied for a long time, to the best of our knowledge all existing solutions assume that the faults at hand are limited in type (e.g., crash), in space (e.g., there is a bound on the number of nodes where faults have occurred), or in time (e.g., at a node a fault may occur only for a finite amount of time). In this paper, we focus our attention on dealing with faults that are of arbitrary type and are unlimited in both space and time. This includes, for instance, dealing with a system where any number of nodes may behave in a Byzantine manner for any length of time, where arbitrary topology changes may occur, where massive unanticipated state corruptions may result, etc. More specifically, we define what it means for a system to locally deal with arbitrary, unbounded faults. Corresponding to (I) and (II), our definition consists of two parts: *strict containment* and *strict tolerance*.

Intuitively speaking, strict containment means that each node in a distributed system behaves correctly as long as no fault occurs at any node within some bounded distance l from it. Said another way, if an arbitrary fault occurs at a node, then the only nodes that can be affected by the fault are less than l distant from it. Note that the definition makes no assumptions about how many nodes may be faulty or how close/far they may be.

Strict tolerance means that for each non-faulty node that is in a fault-affected region, a desired tolerance holds as long as no *more* faults occur at any node within some bounded distance l from it. Said another way, despite the occurrence of faults in any number and at any time at a distance of at least l away from the node, the node satisfies its tolerance property. It is often the case that strict tolerance implies strict containment. However, it is possible that the lower bound for strict containment can be smaller than the lower bound for strict tolerance. In other words, the demand of strict tolerance may be satisfied only with additional cost.

One special case of the notion of strict tolerance is *strict stabilization*, in which starting from an arbitrary system state, a process eventually resumes correct behavior as long as no faults occur within a distance of l. Although we can readily define many other special cases of strict tolerance, in this paper the only tolerance we consider in our examples is that of strict stabilization. One point is worthy of note in this regard: in the traditional study of self-stabilization [9], it is assumed that faults cannot occur infinitely. However, the generalization to strict stabilization has the feature that correct behavior is restored even if faults happen infinitely outside of the locality of interest.

Contributions of the paper. First, we define the notion of strict containment, and establish its relationship with the notion of fault containment with respect to a particular fault class. We also define the notions of strict tolerance and strict stabilization.

Second, we present the first known distributed programs that satisfy strict containment. Our programs solve the problems of graph coloring and dining philosophers. Also, they enjoy the strict tolerance property of strict stabilization.

Third, we identify a necessary condition for a problem to have a solution that has the property of strict containment. On the basis of this condition we derive lower bounds for the problems of graph coloring and dining philosophers. These lower bounds demonstrate that the programs we present are optimal with respect to the size of the locality. We also show that there is no constant bound for the problem of routing, thus showing that this problem does not admit local containment in any solution.

Related work. Fault containment [15] is a well studied design concept in digital systems in general and distributed systems in particular. Any program which prevents faults of a particular class from affecting the correctness of the rest of the system is said to be fault containing. Byzantine agreement programs [13] are but one example.

A number of papers discuss programs that combine fault containment and self-stabilization. Some of these programs stabilize in a local manner, by virtue of assumptions made on the fault type or space-time extent. Several deal specifically with transient faults, and are alternatively referred to as *fault-local* or *time-adaptive* [6, 7, 8, 12]. In other work, Beauquier and Kekkonen-Moneta [3, 4] and Masuzawa [14] study the combination of fault containment with respect to 1-crash and self-stabilization with respect to transient faults. Arora and Kulkarni [11] present an agreement protocol that is fault containing with respect to a bounded number of Byzantine faults and is self-stabilizing with respect to the combination of transient and bounded Byzantine faults.

The problem of dining philosophers is known to have a few solutions that deal with unbounded number of process crashes [5, 17, 18]. Previously [16] the authors extend such a solution so as to be self-stabilizing with respect to transient faults. Yet, the stabilization is not strict.

Organization of the paper. In Section 2 we describe the execution model to be considered. In Section 3 we define fault containment and tolerance classes, explore the relationship between them and identify example problems for each class. In Section 4 we present solutions to graph coloring and dining philosophers problems and prove that they are strictly fault containing and strictly stabilizing. In Section 5 we demonstrate that these solutions are optimal with respect to the size of their fault containment locality. In Section 6 we summarize the results of this paper and discuss possible extensions of this research.

2 Model: Systems, Problems, Programs & Faults

In this section, we recall well-known terminology that we need to define containment and tolerance and discuss demonstrative programs later in the paper.

Systems, processes, and states. A *system* consists of a set of *processes* and a binary reflexive symmetric relation between them, namely the *neighbor relation*. If a pair of processes p and q are in this relation then p and q are *neighbors*. The *distance* between two processes p and q in the system is the length of the shortest path between p and q. The *r-neighborhood* of a process p, denoted $\Gamma(p, r)$, is the set of processes whose distance from p is no more than r.

Each process contains a set of variables. The variables of a process are *local* to it. We let $v.p$ denote a local variable v of process p. Variables range over a fixed domain such as booleans, integers, or other well known types. A *state* of the system is an assignment of a value to every variable of each process of the system from its corresponding domain. Some variables are *input* variables. Some others are *output* variables. Input and output variables are *external* variables.

Programs. A *program* consists of a set of actions for each process of the system and a set of states of the system. These states are the *initial states* of the program. An action has the form: $\langle name \rangle : \langle guard \rangle \longrightarrow \langle assignment\ statement \rangle$. A *guard* is a state predicate over local and state predicates. An *assignment statement* can be simple or quantified. A quantified assignment statement has the form:

$$(\| \langle range \rangle : \langle simple\ assignment\ statement \rangle).$$

A simple assignment statement has a local variable on the right-hand side and an expression involving local and neighbor variables on the left. An input variable can only be on the left-hand side of an assignment statement.

Let $\mathcal{P}$ be a program. An action of a process in $\mathcal{P}$ is *enabled* in a state if the guard of this action is **true** in this state. A *computation* of $\mathcal{P}$ is a maximal fair sequence of states of all variables of the program such that for each state s_i the state s_{i+1} is obtained by executing the command of an action that is enabled at s_i. The execution of an action in a computation is *step*. Maximality of a computation means that the computation is either infinite or it terminates in a state where none of the actions is enabled. Unless stated otherwise we assume that the computation starts from the initial state of the program. We assume that action execution in a computation is *weakly fair*. That is, if an action is enabled in all but finitely many states of an infinite computation then this action is executed infinitely often. The semantics of $\mathcal{P}$ are thus captured by all computations of $\mathcal{P}$ that start from its initial states.

Problems. A *specification* for a process p defines a set of state sequences for this process. These sequences are formed by variables of some subset of processes of the system. This subset always includes p. A computation satisfies a process specification if the projection of the computation on the variables mentioned in the specification is in the specification.

$\mathcal{P}$ *satisfies* specification $\mathcal{S}$ if any computation of $\mathcal{P}$ starting from the initial state of $\mathcal{P}$ satisfies the specification. A *problem* assigns specifications for each process of the system. $\mathcal{P}$ solves problem $\mathcal{Q}$ if $\mathcal{P}$ satisfies the specification of $\mathcal{Q}$ for each process in the system.

Predicates and invariants. A state predicate is a boolean expression over the state of the system. A state *conforms* to a predicate if this predicate is **true** in this state; otherwise the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**.

Let R and S be predicates on system states. An action *ac preserves* R if *ac* is either not enabled in states conforming to R or the execution of *ac* in a state conforming to R produces a state also conforming to R. Let $\mathcal{A}$ be a set of actions. Predicate R is *closed* in $\mathcal{A}$ if all actions of $\mathcal{A}$ preserve R. R converges to S in $\mathcal{A}$ if both R and S are closed in $\mathcal{A}$ and every computation that starts in a state conforming to R contains a state conforming to S.

An *invariant* for a certain process p in program $\mathcal{P}$, denoted $I.p$, is a predicate that is closed in the set of actions of $\mathcal{P}$ such that every computation of $\mathcal{P}$ that starts in a state conforming to $I.p$ satisfies the specification for p. Note that the process invariant does not have to mention local variables of this process only, such invariant may potentially mention the variables of all processes in the system.

Observe that if a program $\mathcal{P}$ solves a problem $\mathcal{Q}$ then for each process p of the system there is an invariant $I.p$ such that each initial state of $\mathcal{P}$ conforms to every $I.p$. Notice that a computation can start from a state that conforms to some process invariants and violates the others. In this case only the particular process specifications are be satisfied.

Faults. The faults that a program is subject to are systematically represented by actions [1]. By way of example, actions may capture faults that are: (i) *transient*, if they simply corrupt the state of a process. (ii) *crashes*, if they disable all actions of a process. (iii) *Byzantine*, if they enable a process to execute actions that arbitrarily change the state of that process. A fault class is a set of fault actions. We let $\mathcal{TF}$, $\mathcal{CF}$ and $\mathcal{BF}$ denote the classes of transient, crash, and Byzantine faults at all processes respectively.

Intuitively speaking, execution of a program in the presence of a fault class corresponds to an interleaving of program and the fault actions. Let $\mathcal{F}$ denote a fault class. Formally, $\mathcal{P}$ *in the presence of* $\mathcal{F}$ is the set of all computations starting from initial states of $\mathcal{P}$ where actions of $\mathcal{P}$ and $\mathcal{F}$ are interleaved, but weak fairness is ensured only for actions of $\mathcal{P}$. In other words, actions of $\mathcal{F}$ may never occur in some computations, may occur only finitely in other computations, and may occur infinitely in still other computations.

Since actions in $\mathcal{F}$ may only directly affect the local state of some processes of $\mathcal{P}$ in a computation c of $\mathcal{P}$ in the presence of F, only some processes of $\mathcal{P}$ may be subject to faults in c, depending on which actions of $\mathcal{F}$ executed in c. The *diameter* of faults in c is therefore the maximum distance between processes all of which are subject to faults in c. $\mathcal{F}$ is *d-bounded*, denoted $\mathcal{F}.d$, for some constant d if the maximum diameter of faults in all computations of $\mathcal{P}$ in the presence of $\mathcal{F}$ is at most d. Otherwise $\mathcal{F}$ is *unbounded*, denoted $\mathcal{F}.\omega$.

3 Fault Containment and Tolerance

We are now ready to define the notions of strict containment and tolerance. Throughout this section let $\mathcal{P}$, $\mathcal{F}$, and p respectively be a program, a fault class, and a process.

3.1 Fault Containment Classes

Definition 1 (Fault containment) *$\mathcal{P}$ is $\mathcal{F}$-*containing *if there is a constant l such that every computation c of $\mathcal{P}$ in the presence of $\mathcal{F}$ satisfies the following condition:*

> *Every process p satisfies its specification in c provided that the distance from p to any process where an action of $\mathcal{F}$ is in c is greater than l.*

The constant l is the *containment radius* of $\mathcal{P}$ with respect to $\mathcal{F}$. We may classify programs on the basis of the fault classes they contain.

Bounded faults. An example of a program that contains a bounded fault class is a leader election program due to Ghosh and Gupta [6]. This program operates on oriented rings. The program locally tolerates transient faults that arbitrarily corrupt the state of a single process. The faulty process may cause its one downstream neighbor to behave incorrectly. Thus, every process whose distance to the faulty process is greater than 1 satisfies its specification. Hence this program is $\mathcal{TF}.0$-containing with locality radius of 1.

Unbounded faults. By lifting the restriction on the extent of the fault we arrive at a program class with stronger containment properties. A few solutions [5, 17, 18] to dining philosophers do not place a bound on the number of crash faults they allow. In each case it is proven that every process whose distance to any crashed process is greater than 2 behaves correctly. That is such solutions are $\mathcal{CF}.\omega$-containing with locality radius of 2.

An unbounded fault class subsumes bounded subclasses. Hence the following proposition.

Proposition 1 *If $\mathcal{P}$ is $\mathcal{F}.\omega$-containing then $\mathcal{P}$ is $\mathcal{F}.d$-containing for every finite d.*

Unbounded Byzantine faults. If $\mathcal{P}$ is $\mathcal{BF}.\omega$-containing it is *strictly fault containing*. Since Byzantine faults allow arbitrary behavior at a faulty process, they subsume all other types of faults.

Proposition 2 *If $\mathcal{P}$ is strictly fault containing, it is $\mathcal{F}$-containing for arbitrary $\mathcal{F}$.*

We find it convenient to reason about the program properties using program states and state predicates [1]. The following proposition aids us in further discussion of strict fault containment.

Proposition 3 *$\mathcal{P}$ is strictly fault containing if there exits a constant l such that for each process p there exists an invariant $I.p$ which is closed with respect to actions of $\mathcal{BF}$ in processes whose distance to p is greater than l.*

Intuitively, Proposition 3 states that the correctness of p cannot be violated by arbitrary actions of processes outside p's locality. To put it another way, the processes in p's locality act as a damper that isolates p from incorrect behavior of other processes. Observe that due to Proposition 3, $I.p$ can mention in a non-trivial manner only the processes whose distance to p is no more than l.

3.2 Strict Tolerance and Strict Stabilization

The fault containment property of a program restricts the spread of fault propagation in the system but does not specify any tolerance properties inside the affected region.

To define tolerance inside such a region we need to consider two kinds of specifications. An *ideal* specification $\mathcal{IS}$ defines the state sequences for a process in case no faults occur in its locality. A *tolerant* specification $\mathcal{TS}$ specifies the (possibly less restrictive) behavior of a process in the presence of faults of class $\mathcal{F}$. We define strict tolerance as follows (Cf. [2]).

Definition 2 (Strict tolerance) *$\mathcal{P}$ is* strictly $\mathcal{F}$-tolerant *if there exists a constant l such that every computation c of $\mathcal{P}$ such that the distance from p to any process subject to faults of class $\mathcal{BF}$ is greater than l satisfies the following condition:*

> *If a process whose distance to p is less than l executes an action of $\mathcal{F}$ in c then p satisfies $\mathcal{TS}$ after all faults of $\mathcal{F}$ stop; p satisfies $\mathcal{IS}$ otherwise.*

From the Definitions 1 and 2 we conclude.

Theorem 1 *$\mathcal{P}$ is strictly tolerant only if it is strictly fault containing.*

We concentrate a particular strict tolerance property of strict stabilization. In strict stabilization the tolerant specification requires that in case the faults in the locality of p stop, p eventually satisfies the ideal specification. Stabilization is usually defined in terms of starting from arbitrary initial state (which is produced by faults). We state our definition in the same vein.

Definition 3 (Strict stabilization) *$\mathcal{P}$ is* strictly stabilizing *if there exists a constant l such that any computation that starts from an arbitrary state contains a suffix where the specification for every process p is satisfied provided the distance from p to any process subject to faults of class $\mathcal{BF}$ is greater than l.*

The constant l is *stabilization radius.* Just as we do for strict fault containment (Cf. Proposition 3) we state an invariant-based necessary condition for a program to be both strictly stabilizing and strictly fault containing.

Proposition 4 *A strictly fault containing program is also strictly stabilizing if for each process p the program stabilizes to p's invariant regardless of the actions of the processes whose distance to p is greater than containment radius.*

4 Program examples

4.1 Vertex Coloring

The vertex coloring of the system is an assignment of colors to the processes such that no two adjacent processes have the same color. Such state of the system is *legal.* We consider the topologies where the vertex degree of the

```
        process p
        const  d : integer
        var  color.p : integer
        *[
recolor :   (∃q : q ∈ Γ(p,1) : color.q = color.p) ⟶
                 color.p = available_color(Γ(p,1), d+1)
         ]
```

Figure 1: Vertex coloring program $\mathcal{PVC}$

graph formed by the system is no more than a constant d and the number of available colors is $d+1$. Each process p has a single output variable *color.p* which holds the color of p. The vertex coloring problem $\mathcal{VC}$ has the following the specification for each process p of the system. Given the system topology, each process p eventually selects a color such that none of its neighbors have the same color.

Consider the vertex coloring program $\mathcal{PVC}$ shown in Figure 1. If a process p shares the color with one if its neighbors, action *recolor* is enabled. When p executes *recolor*, it finds a new color among the available ones and changes its color.

Before we discuss the correctness of the program we prove a couple of properties of the legal states of the system.

Lemma 1 *In the vertex coloring problem there is at least one legal state for every system topology.*

This lemma is proven by a straightforward induction on the number of processes in the system.

Lemma 2 *Given an arbitrary choice of (a) topology of the system, (b) legal state s_1, (c) process p, and (d) color c_2, there always exists another legal state s_2 where p is colored in c_2 and the colors of all processes except p and its neighbors are the same as in s_1.*

Proof: Let the color of p in s_1 be c_1. Let us consider p's neighbors in s_1. If none of them is colored in c_2 then s_2 can be formed by recoloring p in c_2. Let us consider the case where some of p's neighbors are colored in c_2. These neighbors have to be recolored in s_2. Let q be one such neighbor of p. If the degree of q is less than d then there is an available color besides c_2

that q can use in s_2. Suppose the degree of q is d. If some of q's neighbors share a color then again there is an available color in which q can be colored. If all q's neighbors have different colors then none of the neighbors besides p is colored in c_1. In this case the colors of q and p can be swapped in s_2 and q will not have a neighbor with a matching color. The same reasoning applies to every neighbor of p that has color c_2 in s_1. In summary, every neighbor q of p whose color in s_1 is c_2 we can either color in c_1 or in another color such that both the legality of the state is preserved and p is colored in c_2. The lemma follows. □

Let $I.p$ be the predicate

$$(\forall q : q \in \Gamma(p, 1) : color.q \neq color.p)$$

Lemma 3 *$I.p$ is an invariant for program $\mathcal{PVC}$ that is closed with respect to arbitrary actions at processes that do not belong to $\Gamma(p, 1)$. Also, regardless of the executions of the actions of these processes, $\mathcal{PVC}$ stabilizes to $I.p$.*

Proof: Since $I.p$ mentions the variables of p and its neighbors only, it is trivially closed with respect to arbitrary actions of the processes outside the neighborhood of p. Let us show that $I.p$ is indeed an invariant. That is, it is closed with respect to program actions of $\Gamma(p, 1)$ and it guarantees that p conforms to the the specification.

When $I.p$ holds, the action of p is disabled and cannot violate $I.p$. When p's neighbor q executes its action, the color that q selects is different from p's color. By Lemma 2, q can always select such a color. Thus, $I.p$ is closed. When $I.p$ holds, p and its neighbors have no matching colors. Also, since *recolor* is disabled when $I.p$ holds there are only finitely many steps of p in any computation starting from a state conforming to $I.p$, namely none. Hence, $I.p$ is an invariant.

To prove stabilization we observe that when $I.p$ is not satisfied *recolor* is enabled at p. When *recolor* is executed $I.p$ holds. □

The theorem below follows from Lemma 3 and Propositions 3 and 4.

Theorem 2 *Program $\mathcal{PVC}$ is strictly fault containing and strictly stabilizing.*

4.2 Dining Philosophers

In the dining philosophers problem the processes (philosophers) are thinking but periodically request to eat. The problem requires to arrange the eating of the processes such that the following two properties are satisfied for each process p: *safety* – p does not eat in a state where one of its neighbors eats, *liveness* – if p requests to eat, this request is eventually satisfied. It is assumed that a process requests to eat until it starts eating and eventually stops requesting when it eats.

Consider the dining philosophers program $\mathcal{PDP}$ shown in Figure 2. Every process p maintains a boolean input variable $request.p$. Process p *requests* to eat when $request.p$ evaluates to **true**. Variable $state.p$ indicates whether p is thinking (T), hungry (H) or eating (E). For the simplicity of presentation we assume that each neighbor processes p and q share a variable $priority.p.q$ (which is the same as $priority.q.p$). This variable indicates the order of eating for p and q. Both processes can read the variable and each process can give the priority to its neighbor. Process q is a *parent* of p if $priority.p.q = q$. Process q is a *child* of p otherwise.

Each p contains three actions. Action $join$ is enabled when p is hungry and all its parents are thinking. This action allows p to become hungry. When p is hungry and none of its neighbors are eating p may start to eat by executing $enter$. When p finishes eating it executes $exit$. When executing this action p starts to think and gives priority to all its neighbors.

The most involved part of the proof of correctness of $\mathcal{PDP}$ is demonstrating that it satisfies the liveness condition of $\mathcal{DP}$. Hence, we start our proof with Lemmas 4, 6 and 5 which establish the liveness properties of $\mathcal{PDP}$.

Lemma 4 *If an eating process does not execute actions of $\mathcal{BF}$, then it eventually starts to think.*

Lemma 5 *If a process p is hungry and no process in $\Gamma(p, 1)$ executes actions of $\mathcal{BF}$, then p eventually eats.*

Proof: Let p be a correct hungry process. To eat p executes *enter*. Action *enter* is enabled unless there is an eating neighbor. Thus, eventually either p or one of its neighbors q eats. Since q does not execute fault actions, due to Lemma 4, q eventually stops eating. Note that after eating, q becomes p's thinking child. As such q cannot become hungry, and subsequently eating, until p eats.

Since the number of p's neighbors is finite, eventually either p eats or its every neighbor becomes q's thinking child. In the latter case *enter* remains enabled in p until it is executed. $\Box$

process p
var

$request.p$: boolean input,
$state.p : \{T, H, E\}$,
$q : \Gamma(p, 1)$,
$(\forall q :: priority.p.q \equiv priority.q.p : \{p, q\})$

$*[$

join : $request.p \wedge$
$(state.p = T) \ \wedge \ (\forall q : priority.p.q = q : state.q = T) \longrightarrow$
$state.p := H$

$[]$

enter : $(state.p = H) \ \wedge \ (\forall q :: state.q \neq E) \longrightarrow$
$state.p := E$

$[]$

exit : $(state.p = E) \longrightarrow$
$state.p := T,$
$(\| q :: priority.p.q := q)$

$]$

Figure 2: Dining philosophers program $\mathcal{PDP}$

Lemma 6 *If a process p is a thinking process that requests to eat and no process in $\Gamma(p, 2)$ executes actions of $\mathcal{BF}$, then p eventually becomes hungry.*

Proof: To become hungry p has to execute *join*. Action *join* in can be disabled in p only if p has a non-thinking parent. Note that the conditions of Lemma 5 apply to every hungry parent q of p. Thus, q eventually starts to eat. If q is eating, the conditions of Lemma 4 apply to it which means that q eventually starts to think. Note that after eating q stops being p's parent. Note also that none of p's children becomes its parent until p eats. Thus, eventually either p becomes hungry or it loses all parents. In the latter case *join* remains enabled in p until this action is executed. $\square$

Let $I.p$ be the predicate

$(\forall q : q \in \Gamma(p, 1) : (state.p \neq E) \vee (state.q \neq E))$ and
none of the processes in $\Gamma(p, 2)$ are faulty

Lemma 7 *$I.p$ is an invariant for program $\mathcal{PDP}$ that is closed with respect to arbitrary actions at processes that do not belong to $\Gamma(p, 2)$. Also, regardless of the execution of the actions of these processes, $\mathcal{PVC}$ stabilizes to $I.p$.*

Proof: Since $I.p$ does not mention processes outside $\Gamma(p, 2)$ it is trivially closed with respect to their arbitrary actions. To prove that $I.p$ is an invariant we need to show that it is closed under program actions of processes in $\Gamma(p, 2)$ and if it holds then p satisfies the specification of $\mathcal{DP}$. The only program action that can violate $I.p$ is *enter* in a process in $\Gamma(p, 1)$. To violate $I.p$ the process has to execute *enter* when its neighbor is eating. However, *enter* is disabled in this case. If $I.p$ is an invariant p has to satisfy the specification of $\mathcal{DP}$ when $I.p$ holds. The satisfaction of the safety property follows directly from $I.p$. Lemmas 4, 5 and 6 confirm that liveness is also satisfied.

Let us now address the stabilization part of the lemma. Notice that when $I.p$ is not satisfied *leave* is enabled in p. When *leave* is executed $I.p$ holds. Hence $\mathcal{PDP}$ stabilizes to $I.p$ in the absence of faults in $\Gamma(p, 2)$ $\Box$

The following theorem follows from Lemma 7 and Propositions 3 and 4.

Theorem 3 *Program $\mathcal{PDP}$ is strictly fault containing and strictly stabilizing.*

5 Limits of Fault Containment

Definition 4 *A specification $\mathcal{S}$ is r-restrictive if there is a process p with the following property. For every allowed state sequence σ of the specification for p there are two states s_1 and s_2 such that (a) the specification for p of $\mathcal{S}$ requires that s_1 is present in σ and s_2 is absent from it, and (b) s_1 and s_2 differ in the values of output variables of processes whose distance from p is at least r.*

Intuitively, Definition 4 implies that the problem specification prohibits a certain combination of states for processes in the distance of r.

Theorem 4 *If an r-restrictive problem has a strictly fault containing solution with containment radius l then $r \leq l$.*

Proof: Let $\mathcal{S}$ and $\mathcal{P}$ be respectively a problem and a solution conforming to the conditions of the theorem. Let also p be a process conforming to the conditions of Definition 4. Consider a computation of $\mathcal{P}$ that satisfies the specification of $\mathcal{S}$ for p. Let states s_1 and s_2 be as described in Definition 4. Notice that this computation has to include s_1 but cannot include s_2. However, s_1 and s_2 differ only in values of output variables of processes r away from p. By the definition of strict fault containment, $\mathcal{P}$ satisfies the specification of p regardless of the actions of processes more than l away from p. However, if l is less than r then the actions of processes r away from p may change state s_1 into s_2 and violate the specification. Hence l is no less than r. $\square$

Notice that both $\mathcal{VC}$ and $\mathcal{DP}$ are 1-restrictive. Indeed, $\mathcal{VC}$ does not allow two neighbor processes to have the same color and $\mathcal{DP}$ does not allow two neighbors two eat in the same state. Hence the following corollary.

Corollary 1 *The locality radius of a strictly fault containing solution to $\mathcal{VC}$ or $\mathcal{DP}$ is at least 1.*

However, a stronger bound for $\mathcal{DP}$ can be derived from the result by Choy and Singh [5].

Corollary 2 *The locality radius of a strictly fault containing solution to $\mathcal{DP}$ is at least 2.*

Some problems do not have strictly fault containing solutions altogether. Consider routing configuration construction problem $\mathcal{RC}$. Each process p contains an output variable *parent.p*. Variable *parent.p* holds the identifier of one of p's neighbors. This variable of a distinguished process p_1 holds its own identifier. Thus, *parent* variables form a directed graph. $\mathcal{RC}$ has the following specification for each process p. Given the system topology, processes select *parent* such that there is a directed path from p to p_1. Observe that this specification is r-restrictive for arbitrary r. Hence the following corollary.

Corollary 3 *Problem $\mathcal{RC}$ does not have a strictly fault containing solution.*

6 Conclusion

For systems to scale effectively, it becomes increasingly necessary to consider how information flows between system processes so as to handle faults. In this paper, we have built upon previous research that designed distributed programs which were able to contain faults within some locality of the fault. Our approach is to exploit the lower bound limits in the information propagation inherent in the problem itself, and to concentrate on programs whose information propagation approaches these limits. Such programs are amenable to strict containment and strict tolerance, because correct computation in a locality happens regardless of the computation or the faults in the other parts of the system. Besides being of practical interest by itself, this complementary approach provides the limits to traditional fault containment.

In the examples of strict containment and tolerance that we provide, the containment radius was a small constant (1 for vertex coloring and 2 for dining philosophers). However, this is just a peculiarity of the examples we choose rather then a limitation of the concepts themselves. We have examples of programs where the containment radius is an arbitrary k. In one such example, the program maintains a low weight spanner in the Euclidean setting (assuming that the processes in the graph are points in space and the edges are line segments connecting pairs of points). The spanner maintained includes the minimal weight spanning tree, it does not contain short cycles, and each process has a constant number of communication partners. This example program has immediate application for the problem of routing in large scale sensor networks [10]. Due to space restrictions we withheld the detailed discussion of this example from the paper but we intend to discuss this and other examples in our future publications.

Our definitions can be readily extended in various dimensions. For one, instances of strict tolerance other than strict stabilization clearly exist; e.g., one can define strict fail-safe tolerance and strict masking tolerance, analogous to the non-local versions of these definitions in [2]. Strict stabilization itself is readily extended to accommodate faults other than state corruption; specifically, as was suggested in Section 1, one can define the eventual convergence to correct behavior upon starting from an arbitrary system state *only* with respect to non-faulty nodes, provided no more faults occur within a certain distance from the nodes.

This research suggests several opportunities for future work. Forms of fault containment less severe than strict containment need more consideration. In particular, we will focus on programs where fault containment is enabled due to fast correction and recovery, rather than lack of information flow between regions. We will also focus on programs where information propagation is not limited to localities, but is otherwise scalable. We will explore applications of these ideas in applications such as self-configuration of clusters [19], and sensor networks [10], which have provided the bulk of the motivation for the research reported in this paper.

References

[1] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, November 1993. Special Issue on Software Reliability.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerance. *IEEE Transactions on Software Engineering*, 24(1):63–78, 1998.

[3] J. Beauquier and S. Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of Systems Science*, 28(11):1177–1187, 1997.

[4] J. Beauquier and S. Kekkonen-Moneta. On ftss-solvable distributed problems. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 64–79. Carleton University Press, 1997.

[5] M. Choy and A.K. Singh. Localizing failures in distributed synchronization:. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):705–716, July 1996.

[6] S. Ghosh and A. Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, September 1996.

[7] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.

[8] S. Ghosh and S.V. Pemmaraju. Tradeoffs in fault-containing self-stabilization. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 157–169. Carleton University Press, 1997.

[9] T. Herman. A comprehensive bibliography on self-stabilization. *Chicago Journal of Theoretical Computer Science*, working paper, started in November 1996.

[10] J. Hill, R. Szewczyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, November 2000.

[11] S.S. Kulkarni and A. Arora. Compositional design of multitolerant repetitive Byzantine agreement. In *Proceedings of the 18th International Conference on the Foundations of Software Technology and Theoretical Computer Science, Kharagpur, India*, 1997.

[12] S. Kutten and B. Patt-Shamir. Stabilizing time-adaptive protocols. *Theoretical Computer Science*, 220:93–111, 1999.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[14] T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 1.1–1.15, 1995.

[15] V.P. Nelson. Fault-tolerant computing: fundamental concepts. *Computer*, 23(7):19–25, July 1990.

[16] M. Nesterenko and A. Arora. Dining philosophers that tolerate malicious crashes. to appear in proceedings of *22nd International Conference on Distributed Computing Systems (ICDCS 2002).*

[17] P.A.G. Sivilotti, S.M. Pike, and N. Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. In *Proceedings of the 12th IASTED Internation Conference on Parallel and Distributed Computing and Systems*, volume 2, pages 524–529. IASTED/ACTA Press, November 2000.

[18] Y.K. Tsay and R.L. Bagrodia. An algorithm with optimal failure locality for the dining philosophers problem. In G.Tel and P.M.B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG '94*, volume 857 of *Lecture Notes in Computer Science*, pages 296–310, Terschelling, The Netherlands, 29 September–1 October 1994. Springer.

[19] H. Zhang and A. Arora. GS^3: Scalable self-configuration and self-healing in wireless networks. to appear in the proceedings of *Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)* Monterey, CA.